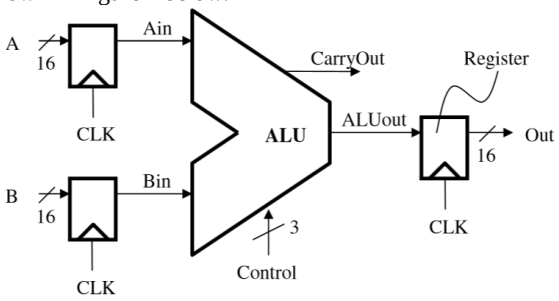
Brian Miler, Cobb Peterson,
Anthony Wash
ECE 3663 – Spring 2014
University of Virginia
<rcp2be, dbm3tz,
adw5da>@virginia.edu

ABSTRACT

Our current client, Portable Instruments Company (PICo), has tasked us with the design and implementation of a proof-of-concept Arithmetic Logic Unit (ALU) for a Digital Signal Processor. This document describes the general approach and justification for our design decisions along with supplemental data that illustrates the ALU's functionality. Throughout the document, we will discuss our process and final design that implements an efficient Arithmetic Logic Unit (ALU) design for the PICo contract.

1. Introduction

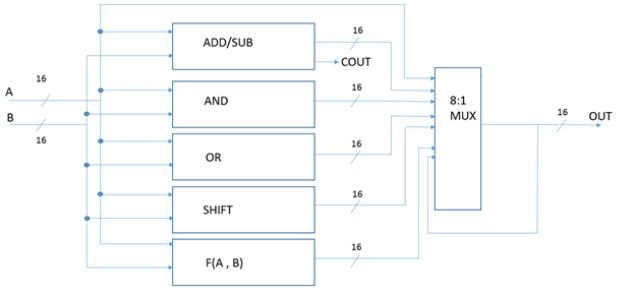
Over the last couple months, our team has been working on designing an ALU for PICo. This ALU desgin is meant to act as a proof-of-concept to convince PICo to hire our design team for their Digital Signal Processor design contract. In specific, PICo wanted our team to implement the Register-Transfer Level design shown in Figure 1 below.



The ALU block shown in Figure 1 must be able to perform a certain function with A and B as specified by the Control input. As per PICo requirements, the ALU must be able to perform bit-wise AND, bit-wise OR, Addition, Subtraction, Shift, Pass A, and NoOp functions as well as an arbitrary function of our choosing. For our arbitrary function, we chose to implement an 8-bit multiplier. The data is loaded into the ALU and leaves the ALU as output by the provided registers. The ALU works efficiently; providing minimal delay, energy, and area.

2. General Overview

Our design process began with a high-level description of the system in terms of each individual function. By connecting the register outputs at A and B to functional blocks and using a Multiplexer at the output, we could perform each function and select the desired output with the Control bits. This approach is illustrated in Figure 2 below.



To implement each function in a timely manner, our design time was split into sections where specific functions were implemented and tested before moving on to the next phase. In our first phase, we implemented the AND, OR and Pass A functions, followed by the Add/Sub, Shift and Register implementations in phase two. In the final phase, we implemented our arbitrary function and worked on design optimization. Each of these phases were tested separately and provide and efficient design that outputs the appropriate function. We reduce power consumption, to better the metric, by only powering the necessary component needed above. The add and subtract function were also integrated to reduce area to better the metric.

3. Implementation

3.1 AND

We used a basic bit-slice approach for our AND block implementation, breaking the block down into bit-wise operations that were then integrated in parallel. For each bit location, the two A and B inputs are fed into a 2 bit AND gate whose output is connected to the corresponding output bit location.

3.2 OR

Again, we used a bit-slice approach to our OR block implementation, using a series of OR gates in parallel. For each bit location, the two A and B inputs are fed into a 2 bit OR gate whose output is connected to the corresponding output bit location.

3.3 ADD/SUB

For the ADD functionality, we decided to implement a ripple-carry adder sequence composed of mirror-adders. The Mirror Adder design was chosen over a conventional static CMOS design to reduce area. Compared to a conventional full adder, the Mirror Adder reduces the transistor count by 4 transistors. For subtraction, we simply used an XOR gate to invert one operand (B in this case) and added one via the carry-in bit. The XOR gate acts as a selective inverter, with the ADD?SUB select bit turning the inverter on or off. The carry-in bit is linked to the ADD/SUB select bit through an inverter. The resulting operation is equivalent to a two's complement binary subtraction

3.4 SHIFT

Our shift function block is composed of multiple 1 bit shift blocks, each of which contains transmission gates that connect the input to the next four outputs. The control logic insures that each shift command is mutually exclusive and that only one set of transmission gates is active at one time. We implemented our shift function using transmission gates as instead of pass gates for one main reason. First, the voltage at the output of the shifter retained its full swing from 0 to VDD. This eliminated the need for buffers to restore the output signal, which would have been needed if we had used pass gates. A disadvantage of using transmission gates was the approximate doubling of the shifter area, but we chose to emphasize full swing output voltage over area. Additionally, if we had used pass gates, the area of the shifter would have been increased by the necessary buffers at the output.

3.5 Registers

We used a parallel combination of 16 D-Flip Flops tied to the Clock for the rising-edge registers seen in Figure 1. The flip flops were built using tri-state inverters and inverters in a positive feedback loop to hold the data. An inverted Clock signal is also sent to each register to power transmission gates within

3.5 8-Bit Multiplier

For our arbitrary function, we decided to implement an 8-Bit Wallace-Tree Multiplier. We felt that a multiplier would complement the functionality provided by the Add and Shift blocks. A Wallace-Tree multiplier was specifically chosen to reduce delay. We chose 8-Bit inputs in order to remain consistent with our 16-Bit output. The Wallace Tree design uses compression to generate partial products that are fed into the multiplier's adder array. The design also contains fewer half and full adders than a conventional shift and add design.

3.6 Pass A

The Pass A function allows whatever signals are present at input A to be directly routed to the ALU output. We accomplished this by connecting input A to a dedicated MUX input with its own control code. The Pass A function is simple to implement and costs no additional area, so it was easy to include in our design.

3.7 16-bit 8:1 Multiplexer

Our multiplexer consists of sixteen 8 to 1 multiplexers that are parallel connected to route the outputs from the function blocks to the output register. Each block can be connected to the output via a specific MUX control code.

4. Optimization

4.1 Power

In an effort to reduce power consumption, we decided to tie each of the power lines of each block to transmission gates such that only one block (i.e. the selected operation) is powered at a time. This reduces power by lowering the number of unnecessary transitions in a given cycle. The tradeoffs for this are increased delay and increased setup and hold times for the control lines.

4.2 Mirror Adder Sizing

We increased the sizes of the transistors in our mirror adder to decrease the delay of the carry chain. Because our adder was a ripple carry design, the carry chain composed the majority of the critical path. Optimizing each adder to be as fast as possible was critical to reducing the carry chain delay and thus the critical path delay. This increase in size was based on Figure 11.6 in the Rabaey textbook. Compared to other adders, the Mirror Adder has a fairly low area when scaled to larger bit inputs and has greater immunity to noise due to its CMOS design.

4.3 Multiplier Layout

By using a Wallace-Tree multiplier instead of an array multiplier, we reduce delay (by lowering the number of carry chains) and area.

4.4 Add/Sub Layout

By using a mux and added carry in-bit, we were able to save space combining the add and subtract functions. The tradeoff here was that we acquired a bit more delay for our Add/Sub functions.

5. Results

In the final design, we used the Control-to-Function layout as shown in Table 1. The below table provides data for each block used. Data is provided for area, delay, power, and energy for the different blocks created. This data is provided for the add, subtract, and, or, shift, multiply, pass a, register, and mux provided. The add and subtract have the same area because they were integrated into the same block to reduce area. This will provide us with a more efficient metric.

Table 2. This Table shows the general metrics for each functional block

Block	Area	Delay (ps)	Power (µW)	Energy (pJ)
ADD	177930n	246.6	469.9	0.94
SUB	177930n	181.00	480.2	0.96
AND	12960n	17.0	24.0	0.048
OR	12960n	16.8	22.8	0.046
SHIFT	43740n	88.0	101.3	0.20
MULT	613440n	138.4	280.3	0.029
PASS A	NA	NA	NA	NA
REGISTER	43200n	13.2	87.2	0.523
MUX	322560n	53.6	125.9	0.256

Table 3. This Table shows the values obtained from each metric measurement.

Delay	Area	Energy	Metric
411.4ps	1.31*10^-3 m	4.945 pJ	2.66986 × 10^-24

6. Conclusion

The ALU we chose to design should be selected for the PICo contract for several reasons. A multiplier is included in the design to facilitate fast multiplication without using the ADD function in a multiply routine. The Wallace Tree design of our multiplier gives lower area and delay compared to traditional multipliers. Our adder and subtractor design incorporated both functions into one logic block, with ADD/SUB being selected via MUX control code. This multifunction circuit allowed us to save power and area in our design. Additionally, we included circuitry to supply power only to the function block that is currently selected, thus avoiding wasting power in unused blocks. To limit the area of our chip, we only increased transistor sizes where absolutely necessary. Except for the ADD/SUB block, most other transistor sizes are the smallest available. Based on our design decisions and optimizations, we have created an ALU that has a small metric with respect to PICo's design criteria.